UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Elaboración de un motor de animación 3D para three.js

David Pedro Chico García
Tutor: Carlos Aguirre Maeso

Marzo 2017

Elaboración de un motor de animación 3D para three.js

AUTOR: David Pedro Chico García
TUTOR: Carlos Aguirre Maeso

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Marzo de 2017

Resumen

La industria de los videojuegos, a la par que el mundo de la informática, ha experimentado una creciente evolución en los últimos años, llegando a convertirse en una de las más grandes del mundo del entretenimiento. Los videojuegos han demostrado ser una potente herramienta de entretenimiento, a la vez que una forma de expresión personal para artistas y creativos totalmente nueva.

Desde mi perspectiva una de las cosas más bonitas que ofrece la informática a la humanidad es una capacidad creativa ilimitada, que depende solo del conocimiento técnico y la creatividad del desarrollador. Puede parecer trivial comparado con las grandes aportaciones de la informática; el internet, la accesibilidad, las facilidades en la gestión. No pretendo judgar cuál es una mayor aportación a la sociedad porque creo que está claro. Pero también está claro que la expresión artística ha acompañado a la humanidad desde sus orígenes y siempre hemos buscado formas más completas de expresar nuestras inquietudes personales.

Con las nuevas facilidades en el desarrollo de videojuegos, los motores de última generación, la industria ha dejado de estar dirigida por unos pocos y ha pasado a las manos de los desarrolladores indie, que aportan una mayor riqueza y pueden llevar a cabo trabajos más personales y arriesgados.

También las tecnologías están cambiando; inicialmente los videojuegos corrian solo en consolas especializadas; ahora parece que el futuro de los videojuegos está en los dispositivos móviles. Con el surgimiento de HTML5 y WebGL se abre una nueva puerta todavía sin explorar.

Este proyecto se plantea como una herramienta más de ayuda para el desarrollo de videojuegos en three.js, un motor de videojuegos que corre sobre WebGL.

Palabras clave

Motor de videojuegos, motor de animación, huesos, claves, esqueleto, modelo, blending, morph, maquina de estados, generación de código, interfaz

Abstract

The videogames industry has evolved drastically for the last few years, same as the computer technology, growing to become one of the biggest entertaining industries. Videogames has probed to be a powerfull entertaining tool, as well as a great facility for personal and artisticall expression

One of the greatest achivements of computer technology, from my point of view, is that it offers humanity a canvas with unlimited creation capacity; the only limit is the technical knowledge and our own creativity. It may seem incidental compared to the biggest achievements of informatics (internet, acessibility and so on). There is no point in comparing witch is a biggest achivement for humanity since, I believe, that is clear. But it is also forthright to say that artistic expression has accompany humanity since it was born and we have always looked for new and more effective ways of expressing ourselfs.

Following the new facilities in videogames development, last generation game engines, the industry is no longer controled by a few companies and, instead, it is now in the hands of indie developers, providing variety and designing more personal and risky projects.

Technology is also evolving and, unlike the first games that used to run only in specific consoles, it seems that the future of videogames is in mobile devices and multiplataform. With the raise of HTML5 and WebGL we have a new and unexplored path ahead of us.

This project pretends to offer a tool to facilitate certain tasks in game developing, more specificaly games developed in three.js, an engine built over WebGL.

Keywords

Game engine, animation engine, bones, keyframes, skeleton, mesh, skeletalmesh, blending, morph, state machine, code genaration, front-end

Agradecimientos

Debo agradecer, en primer lugar, a mis padres, porque sin ellos no estaría aquí. No solo me han criado y dado de comer durante los últimos 22 años, sino que si no fuera por ellos posiblemente no me habría metido a estudiar Ingeniería Informática. Aunque no siempre tuve claro que fuera el camino para mí, se que todo lo que he aprendido en estos años me va a ser útil de cara a mi futuro profesional y me define como persona, y todo eso es, en gran parte, gracias a ellos.

Quiero agradecer, en segundo lugar, a la Asociación Infantil y Juvenil *"El Patio"* y a las personas que la conforman, las cuales me han acompañado durante todos estos años. Sin duda ha sido uno de los apartados más trascendentales de estos años, me ha aportado mucho más de lo que yo, potencialmente, puedo haber ofrecido en el tiempo durante el que he colaborado como monitor.

No quiero dejar de mencionar a la universidad, profesores y compañeros, por la oportunidad que me han brindado de aprender y crecer, a nivel profesional e intelectual, durante estos años de carrera. A mi tutor, Carlos Aguirre, por permitirme hacer un Trabajo de fin de Grado en un ámbito que me apasiona y por acompañarme durante todo el proceso.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

Los videojuegos en **web abierta** son una novedad tecnológica **aún por explotar.** Desde juegos sencillos producidos por desarrolladores independientes hasta juegos con inversiones más importantes, la industria puede encontrar aquí un modelo totalmente nuevo de negocio y distribución. Profundizaremos mucho más en las características que nos ofrecen, sopesando los inconvenientes y las ventajas que ofrece la web y HTML5 como plataformas donde correr nuestros juegos.

Por otro lado las tecnologías y los nuevos motores de videojuegos están haciendo cada vez más **fácil y accesible** la industria a personas con mayores capacidades artísticas y menos preparadas tecnológicamente, permitiendo desarrollar mecánicas sencillas sin tener apenas conocimientos de programación. Estas herramientas permiten despreocuparse de las cuestiones más complejas de bajo nivel para centrarse en la parte más artística o en desarrollar una mecánica divertida. Además facilitan un **desarrollo más rápido** y por lo tanto menos arriesgado, lo cual permite mayor flexibilidad a la hora de innovar y experimentar.

La parte artística de los videojuegos ha pasado a ser uno de los principales protagonistas, siendo en gran parte el motivo de que un videojuego llame la atención. Sigue siendo esencial mantener una mecánica y jugabilidad divertidas, siendo estos los aspectos de mayor peso en el desarrollo de un videojuego - al final no importa lo bien que se vea si es un juego aburrido, demasiado fácil o demasiado difícil. Pero el público es cada vez más exigente con los gráficos y castigarán a todo juego que tenga unos gráficos mediocres o poco llamativos.

1.1 Motivación

Este trabajo de fin de grado pretende ofrecer una ayuda al desarrollador que ofrezca ciertas facilidades en el diseño de videojuegos de web abierta. Concretamente se centra en la parte correspondiente a la animación que es, visualmente, uno de los elementos más importantes. Una animación correcta y funcional hará que el jugador se crea o se integre en el mundo del juego obteniendo una experiencia mucho más intensa. Además, gestionar correctamente las animaciones de los personajes en un videojuego es una labor que tiene un compomente artístico importante, además de la dificultad técnica. Lo ideal sería tener herramientas que puedan **integrar la labor de un animador con la de un programador de forma sencilla**.

Me parece importante destacar que gran parte de mi motivación para realizar este proyecto es personal. Desde siempre he estado interesado en el desarrollo de los videojuegos, tanto por la parte tecnológica como la artística. Creo que esto me da un perfil más que adecuado para diseñar y desarrollar una aplicación de este tipo porque conozco de primera mano las tecnologías y las necesidades de un animador.

Por otro lado es una herramienta que me **resultará últil de cara al futuro**, ofreciendome una infraestructura completa y sencilla para gestionar las animaciones de cualquer proyecto de videojuego que pretenda desarrollar sobre Three.js. Por supuesto, también le será útil a quien quiera que pretenda elaborar un proyecto de estas características.

1.2 Objetivos

El principal objetivo de este trabajo es elaborar un **motor de animación** para three.js, es decir, una **pieza de software** que pueda integrarse en un videojuego desarrollado en three.js y **que proporcione la infraestructura** necesaria **para diseñar y organizar las animaciones** de forma adecuada.

Tomando como principal base de referencia los motores Unity y Unreal 4, la principal forma que tendrá este motor es el de un **gestor de máquinas de estados** (asociadas a los diferentes comportamientos de los assets animables), encargado de que cada modelo esté ejecutando la acción adecuada en cada instante del juego.

Por otro lado es importante diseñar un **front-end o un editor** que haga más accesible el diseño de estas máquinas de estados y ofrezca un workflow adecuado para poder crear e integrar en el videojuego estos assets animados. Este objetivo es secundario en comparación con el motor de animación que es el núcleo del proyecto, pero tiene una trascendencia considerable en el conjunto del proyecto. Como ya hemos explicado es una herramienta orientada a que trabajen con ella personas sin una preparación técnica espacialmente profunda.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Estado del arte:** En esta sección se expondrá una breve introducción a las tecnologías sobre las que se apoya este proyecto; de los videojuegos en web, de WebGL y Three.js y una concisa introducción sobre los conceptos de animación que serán necesarios para entender el proyecto. También se presentarán otras soluciones dadas por motores de animación ya comercializados.

- **Diseño:** Aquí se realizará un análisis de los requisitos del sistema y se presentará el diseño previo realizado, diagramas de clases y de flujo.

- **Desarrollo:** En este capítulo se explicará la solución final desarrollada y las decisiones tomadas durante la codificación.

- **Integración, pruebas y resultados:** Aquí presentaremos las pruebas realizadas para validar el motor y el editor, así como los resultados.

- **Conclusiones y trabajo futuro:** En este capítulo se discutirá sobe las conclusiones que hayan surgido de la elaboración del proyecto así las posibles vías en las que se podría seguir desarrollando el mismo.

2 Estado del arte

2.1 Los juegos de web abierta

Con el avance de las tecnologías la industria de los videojuegos está evolucionando drásticamente. Actualmente la mayor parte de la industria se encuentra en los dispositivos móviles y no en las grandes consolas. Cada vez menos gente va a estar dispuesta a gastarse 60€ en juegos cuya producción a costado millones. Las nuevas generaciones, acostumbradas a contenidos rápidos, que apenas exigen esfuerzo, preferirán las opciones más sencillas, baratas y rápidas que ofrecen los dispositivos móviles.

Pero más allá del tipo de juego, lo que está resultando ser más conveniente es que estos puedan correr en cualquier dispositivo; android, IOS, ordenador e incluso consola, un solo código que pueda correr en todos los navegadores. Aunque aún encontramos algún problema de compatibilidades, esta es la característica más potente de los juegos que corren sobre HTML5.

Sin necesidad de utilizar un plugging externo (como hacía su predecesor *flash*), HTML5 permite gestionar contenidos gráficos interactivos. Los juegos basados en esta tecnología se pueden distribuir al público más grande posible, que son los usuarios de internet, y se puede reproducir en cualquier dispositivo que disponga de un navegador compatible. Esta nueva realidad amenaza con cambiar drásticamente las reglas en la industria de los videojuegos; cualquiera puede distribuir sus contenidos, de forma rápida a todo el mundo.

Aunque estemos ante una realidad novedosa, estas técnicas ya están siendo usadas en el desarrollo de sitios y aplicaciones que corren en la web, demostrando que son fiables y estables, y llevan ya un importante recorrido y evolución desde su origen.

Los juegos en web abierta presentan varias ventajas importantes, algunas de las cuales ya hemos ido hablando.

La facilidad y el poco coste en la distribución a un público masivo es una de las ventajas, sin duda más importantes. El hecho de que sea multiplataformas y que no hara falta, una vez se extienda por todos los navegadores el estandar, instalar ni mantener pluggings adicionales es otro aspecto importante a considerar.

Por otro lado es muy sencillo empezar, hay muy buena documentación y una comunidad muy potente. Además, al desarrollarse en HTML5 y Javascript todo el código se ejecuta en el lado del cliente y es visible para todo el mundo, por lo cual, a la hora de aprender, es fácil encontrar cientos de ejemplos prácticos.

Hay que considerar a su vez las desventajas, las cuales son principalmente dos. Hemos dicho ya que el código es multiplataforma; *"write once, run everywhere"*. Esto es cierto, pero es importante resaltar que de un dispositivo a otro se puede comprometer enormemente el rendimiento. Aunque el código pueda ser interpretado por el navegador de un telefono, si el juego es muy pesado, corriendo sobre WebGL, con miles de polígonos y texturas, puede que el dispositivo no lo aguante.

Por otro lado, aunque lo hemos mencionado entre las ventajas, también es un inconveniente para muchos que el código tenga que ser forzosamente abierto dado que no se compila. Cualquiera puede ver y analizar tu código, incluso copiarlo y reutilizarlo. Para los que contribuyen a la cultura del software libre o para un desarrollador independiente esto no tendría por que suponer ningún problema, pero para una empresa que elabore y venda sus productos, compitiendo con otras, puede ser un inconveniente importante a considerar (e incluso un motivo que haga imposible para las grandes compañias aprovechar estas tecnologías hasta que cambie esta realidad). [1]

2.1.1 WebGL y Three.js

WebGL 2.0 (Web Graphics Library) es una API de Javascript que permite la renderización de gráficos 3D de alta calidad directamente sobre el canvas de HTML5 (en el navegador). Existen otras tecnologías que permiten trabajar con gráficos 2D como SVG y Processing. Esta basado en OpenGL ES 3.0 y tiene un amplio soporte en los navegadores modernos. La solución que ofrece es muy completa y permite trabajar en profundidad con las nuevas tecnologías de los gráficos 3D; shaders GLSL, luces, animación... [2]

La API de WebGL, aunque es muy completa, resulta tremendamente engorrosa de utilizar. Por ello surge three.js que encapsula y simplifica el uso de WebGL ofreciendo una API más amigable y sencilla.

Cabe destacar que WebGL y Three.js no son motores de videojuegos sino que son librerías gráficas que ofrecen una infraestructura de animación y render. Para poder desarrollar un videojuego habrá que completar con otras librerías que se encarguen de gestionar el audio, la entrada desde el ratón o el teclado y las fisicas.

2.2 La animación en los videojuegos. Blend y Morph

2.2.1 La animación 3D

La animación en los videojuegos es una pieza de gran relevancia en la experiencia del usuario. Unas animaciones creíbles harán que el usuario se integre más en el mundo que los desarrolladores pretenden crear.

La animación consiste en realizar una secuencia de imágenes estáticas que, a ojos del observador, generan sensación de movimiento. Estas pueden ser dibujos o, en la animación por ordenador, serían imágenes renderizadas por la GPU o la tarjeta gráfica. El renderizado de una sola imágen puede suponer horas de cálculo, incluso días o meses. Los estudios de animación que renderizan películas enteras utilizan las llamadas granjas de render. En estos casos es interesante recortar en la medida de lo posible el tiempo de

cálculo por cuestiones económicas, pero cuanto más tiempo esté renderizando más calidad tendrá el producto final.

Centrandonos más en lo que nos interesa, que serían los videojuegos o la animación para videojuegos, a diferencia de la animación para películas, el tiempo de render está mucho más limitado teniendo que ser de 60 fotogramas por segundo, o lo que es lo mismo, hay que renderizar la imagen 60 veces por segundo. Por ello la tecnología está mucho más limitada, teniendo que buscar técnicas que reduzcan el número de polígonos o facilitando el cáluclo de las luces con elementos precalculados (lghtmaps).

Todo elemento en un videojuego 3D está formado por un modelo o mesh y unas texturas. Este modelo consiste en un conjunto de aristas, vértices y caras que se utilizan para calcular la imagen que se proyecta en la pantalla en cada momento. Cuando el modelo es estático (StaticMesh) la cuestión termina aquí, pero no pasa lo mismo con los personajes o con los modelos animados (RiggedMesh o SkinnedMesh).

Ya hemos explicado que la animación es una secuencia de imágenes estáticas. Bueno, pues una acción o animación en un modelo sería entonces un conjunto de modelos estáticos que, presentados de forma secuencial, crean la ilusión de que el modelo está vivo.

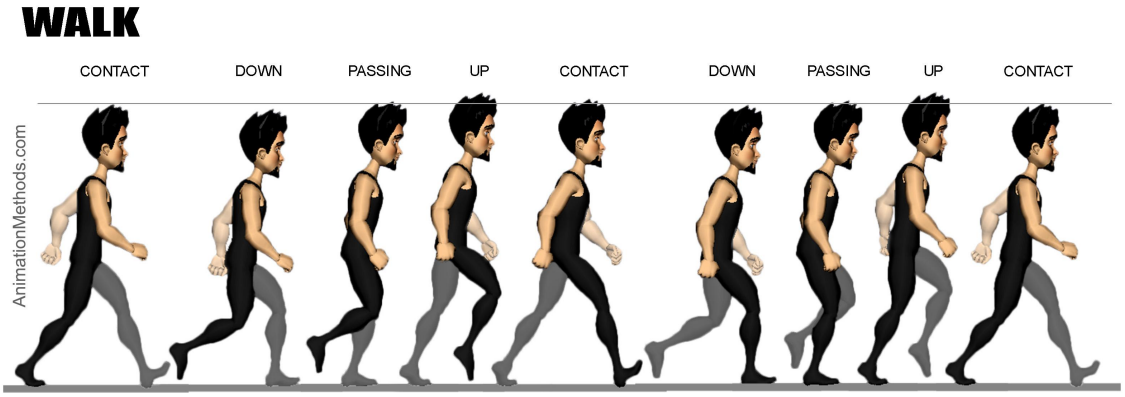


Figura 2- 1: Animación 3D

2.2.2 El rigging y skinning y el proceso del animador

Sería muy complejo diseñar un modelo distinto para cada una de las cientos de poses que puede haber en un segundo de animación. Por ello el proceso de animación pasa por una nueva etapa que sería el rigging y el skinning del objeto.

El **rigging** (aparejo) consiste en diseñar un rig, que consiste en una serie de huesos u objetos que permiten definir transformaciones sobre un objeto concreto. Un rig buen hecho facilita mucho la labor del animador y permite diseñar movimientos naturales. [3]

El **skinning** (cubrir de piel) consiste en asignar a cada uno de los vértices del modelo (que se podría entender como la piel) el grado de influencia que sufre por cada uno de los huesos de un rig dado. [4]

Tras este proceso el objeto pasa a poder ser deformado simplemente transformando los huesos del rig.

La animación 3D es un proceso complejo; en primera instancia hay que diseñar un modelo adecuado para deformarlo, con una topología limpia y flexible. Después, hay que implementar un rigging adecuado; el animador tiene que diseñar un conjunto amplio de movimientos y animaciones que, por último, un artista técnico o programador deberá integrar todo el juego.

Me parece importante destacar que es un proceso con una gran dificultad técnica pero son una **importante componente artística**. De aquí se evidencia la necesidad de tener interfzaces buenas e intutivas adaptadas a las necesidades de los animadores que dificilmente serán ingenieros informáticos.

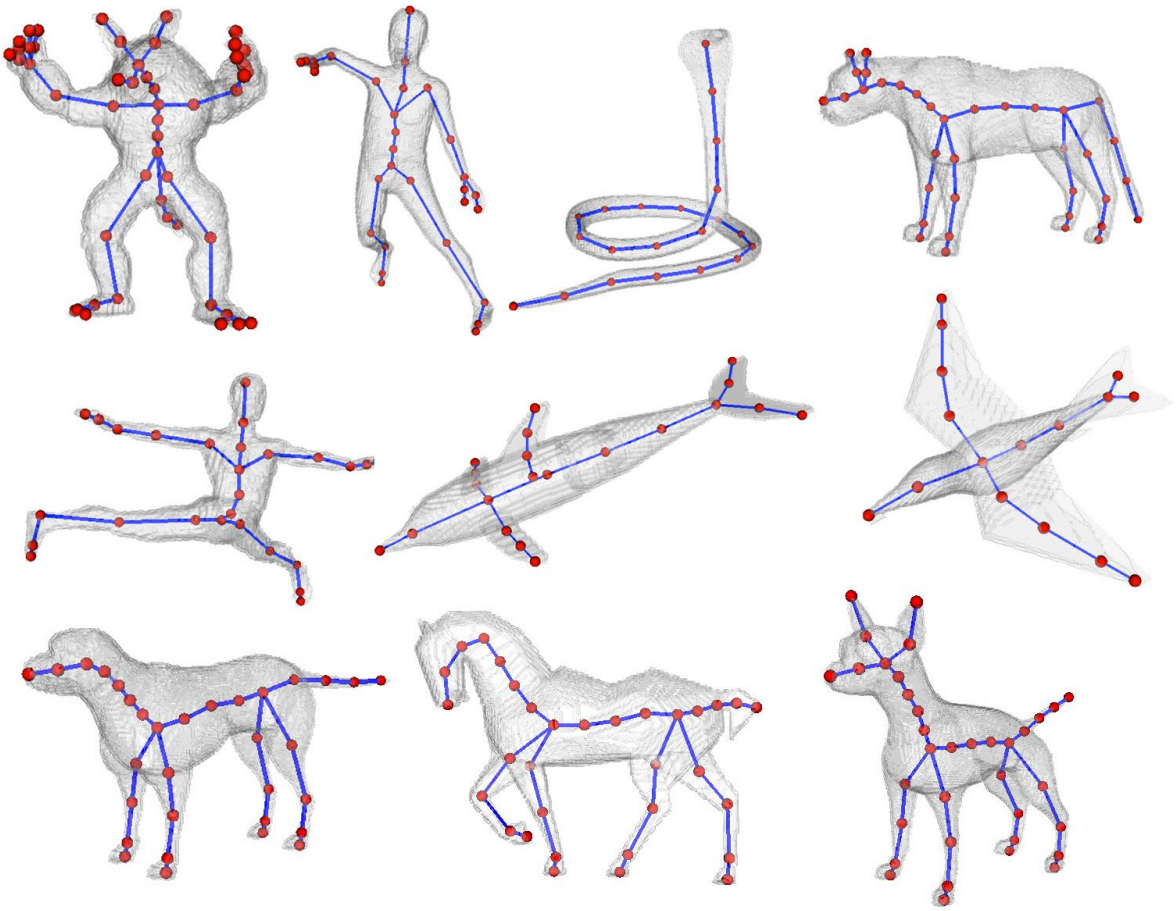


Figura 2- 2: Rigging y Skinning

Al final las animaciones son lo que dan vida a cada uno de los personajes del juego. Al ser un contenido interactivo, se añade la dificultad de que el juego debe saber en cada momento que acción tiene que estar realizando cada personaje. Este problema es del que principalmente se ocupa el motor de animación que integra el videojuego.

Dos conceptos importantes al programar modelos animados para videojuegos son el blend (o blend tree) y el morph.

2.2.3 El blending

Blending [5], que se traduce literalmente como mezclar, consiste exactamente en eso; integrar varias animaciones entre sí. En general esto se hace mediante un sistema de pesos; a cada acción o animación que se está ejecutando concurrentemente en un modelo dado se le asigna un peso concreto que representa la influencia que va a tener sobre el modelo.

Estos pesos se pueden modificar de forma gradual para realizar transiciones suaves entre animaciones, aplicando efectos de *fadein* y *fadeout,* permitiendo que los cambios de una animación a otra sean naturales y poco bruscos.

Otro uso también interesante es el de los *blendtrees*, que son estructuras encargadas de gestionar el blending entre varias animaciones. Esto pude servir para crear una animación más realista que integre diferentes tipos de animaciones. Un ejemplo práctico podría ser integrar las animaciones de andar y correr en función de la velocidad, de tal forma que a medida que aumente la velocidad, aumenta el peso de la animación correr. Si la velocidad aumenta de forma gradual quedaría un efecto mucho más creíble dado que la animación se adaptaría constantemente a la velocidad del personaje. Otra posible integración sería mediante el uso de un vector dirección (que indica la dirección en la que avanza un personaje) conjuntamente con las animaciones de andar hacia delante, atrás, izquierda y derecha. Tendríamos que:

```
Var alpha = angulo(direction)

Forward.weight = if sin(alpha) > 0 : sin(alpha) ? 0
Backward.weight = if sin(alpha) < 0 : -sin(alpha) ? 0
Left.weight = if cos(alpha) > 0 : cos(alpha) ? 0
Right.weight = if cos(alpha) < 0 : -cos(alpha) ? 0
```

Los pesos son función del vector dirección y se tendrían que actualizar cada ciclo de ejecución del juego. Esto permite que en vez de tener 4 animaciones tenemos infinitas combinaciones para cada valor de alpha.

2.2.4 El Morphing

El morphing [6] (mutación en español) es otra técnica que permite mezclar varias animaciones. En el blending varias animaciones se aplicaban a la vez en todos los vértices del modelo. El morphing consiste en decidir que vertices se verán afectados por que animaciones. Esto permite que una parte del modelo esté realizando una acción mientras que otra está realizando otra.

Un ejemplo práctico es el soldado que por un lado esta andando, parado o corriendo de cintura para abajo, y de cintura para arriba está apuntando, disparando o cargando.

De nuevo gracias a esta técnica podemos combinar unas pocas animaciones para generar cientos, adaptandolas a las diferentes necesidades del juego.

2.3 Motores de animación

En esta sección vamos a analizar diferentes motores de animación que están siendo usados en motores potentes como son Unreal 4 y Unity (los dos más extendidos), y por lo tanto sabemos que funcionan.

Estos serviran de referencia para el diseño del motor desarrollado en este TFG.

Figura 2- 3 y 2-4: Unreal 4 y Unity3D

Tanto unity como unreal ofrecen pluggings para correr en web, pero aún son más pesados que usar WebGL directamente.

2.3.1 Unreal 4 Animation System

Unreal Engine es un motor de videojuegos desarrollado por Epic Games y que, ha sido galardonado al premio de "El motor de videojuegos más exitoso" por Guinness World Records [7]. Ha sido utilizado en incontables juegos comerciales de mecánicas y temáticas variadas. La última versión, Unreal 4, ha salido al mercado ofreciendo un uso gratuito lo que ha hecho que se extienda entre desarrolladores independientes. [8]

Unreal4 se basa en el uso de una tecnología propia denominada blueprints, que son objetos programables gráficamente. Las animaciones en unreal se diseñan con un tipo específico de blueprint a partir de gráficos de eventos (un diagrama de secuencia en el que se especifica el flujo de las acciones), diagramas de estados y blendtrees para cada estado. Permite un nivel de control y flexibilidad muy profundo, manejando directamente los huesos desde el propio.

Como principal inconveniente tiene que es complejo y tiene una curva de aprendizaje lenta y un grado de especialización alto para trabajar con el sistema de animación.

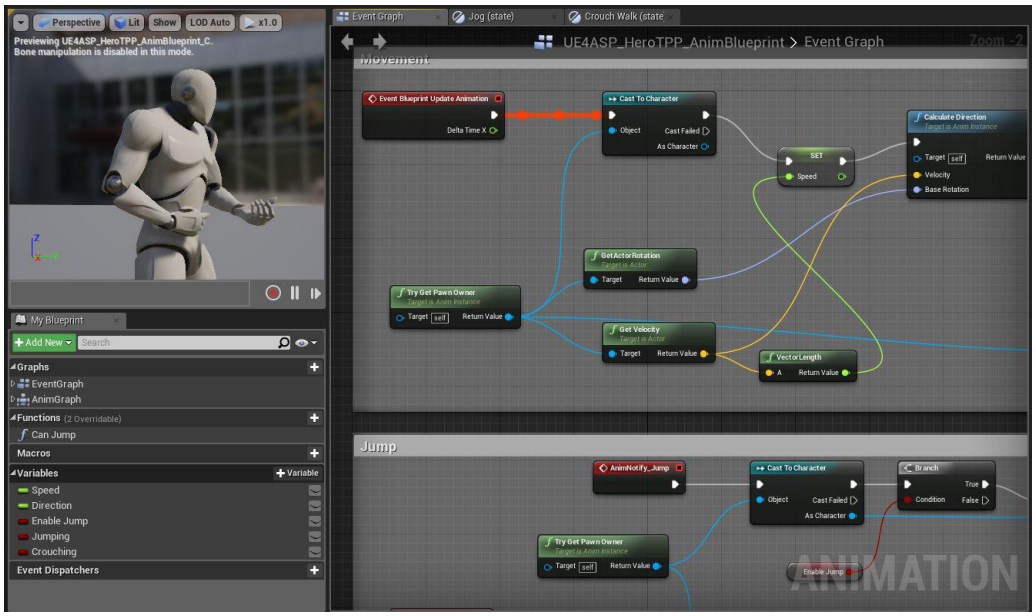


Figura 2- 5: Animation Blueprints en U4

2.3.2 El motor Mechanim de Unity

Las animaciones en Unity están gestionadas por el motor Mechanim, que está fuertemente basado en máquinas de estados. Es mucho más sencillo de aprender que en el caso de Unreal, pero implica un mayor uso de programación en la integración. Cada estado representa una animación o varias combinadas mediante un blendtree. Permite también el uso de capas y máscaras de morphing.

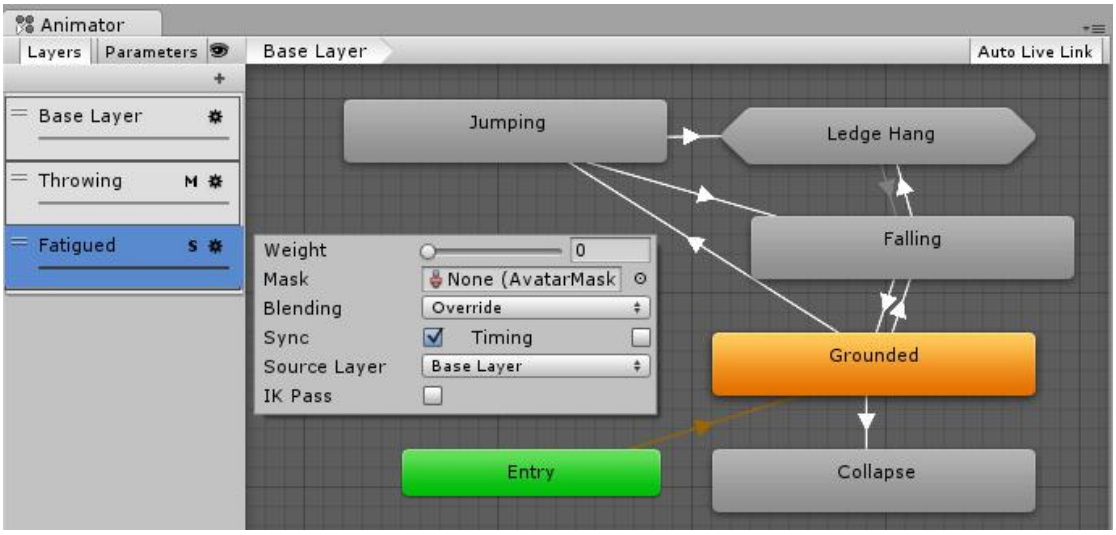


Figura 2- 6: Máquinas de estados en Unity

El motor de este proyecto está fuertemente inspirado en Mechanim.

3 Diseño

3.1 El motor ThreeAction

3.1.1 Analisis de requisitos

La principal función de este motor es gestionar de forma sencilla a la par que flexible las animaciones de un modelo. Mediante conceptos y herramientas sencillas debe permitir gestionar modelos con una gran complejidad de animaciones siguiendo una lógica que facilite la mecánica.

Los requisitos del motor serían los siguientes:

Nº	Requisito
1	Ofrecer una infraestructura que permita programar el motor para cada SkinnedMesh
2	Permitir gestionar varias animaciones de forma organizada
3	Ofrecer una interfaz que permita comunicar el juego con el motor para que este se pueda actualizar en tiempo real atendiendo a las necesidades de la ejecución
4	Proporcionar transiciones adecuadas entre las animaciones
5	Permitir que haya animaciones que puedan ser lanzadas en cualquier momento independientemente del estado del personaje
6	Gestionar la carga del skinnedmesh

3.1.2 Diseño

El motor ThreeAction es, en esencia, un gestor de máquinas de estados. Un personaje sería una máquina de estados finita que va pasando de un estado a otro dependiendo de la interacción con el juego y el jugador. Cada estado representa una acción concreta. Por ello hace uso de un patrón de diseño **composite** que integra muchos estados y transiciones para diseñar una estructura compleja.

Esta formado por 3 piezas fundamentales que definen la máquina, la cual denominaremos como *ActionMachine*. Este ActionMachine puede integrarse como una parte más de un personaje o *character* dentro del juego. Integra y controla un BlendCharacter (que es una de las clases internas de three.js) de tal forma que el programador que utiliza ThreeAction en su juego no deberá preocuparse por los blendings de una acción a otra.

En la figura 3-1 se muestra mediante un diagrama de clases la estructura. El punto que centraliza gran parte de la lógica es la clase *ActionMachine*. Esta es la encargada de, en tiempo de ejecución, verificar cuando se tiene que pasar de un estado a otro. Por otro lado el StateMachine contiene y permite gestionar las variables de control (propiedades) del personaje.

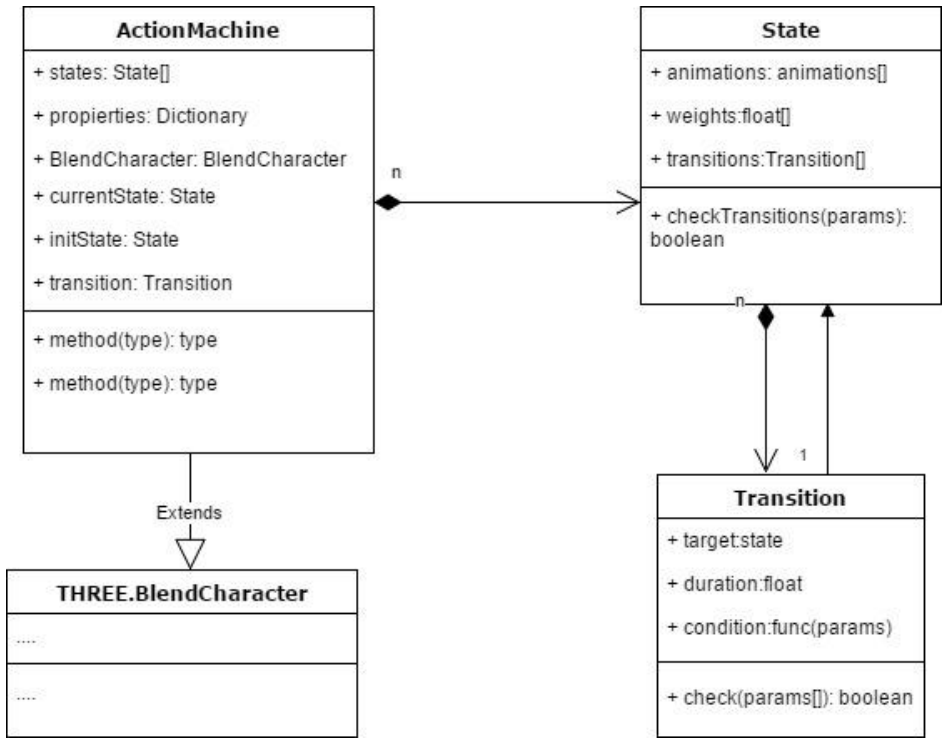


Figura 3- 1: Diagrama de clases

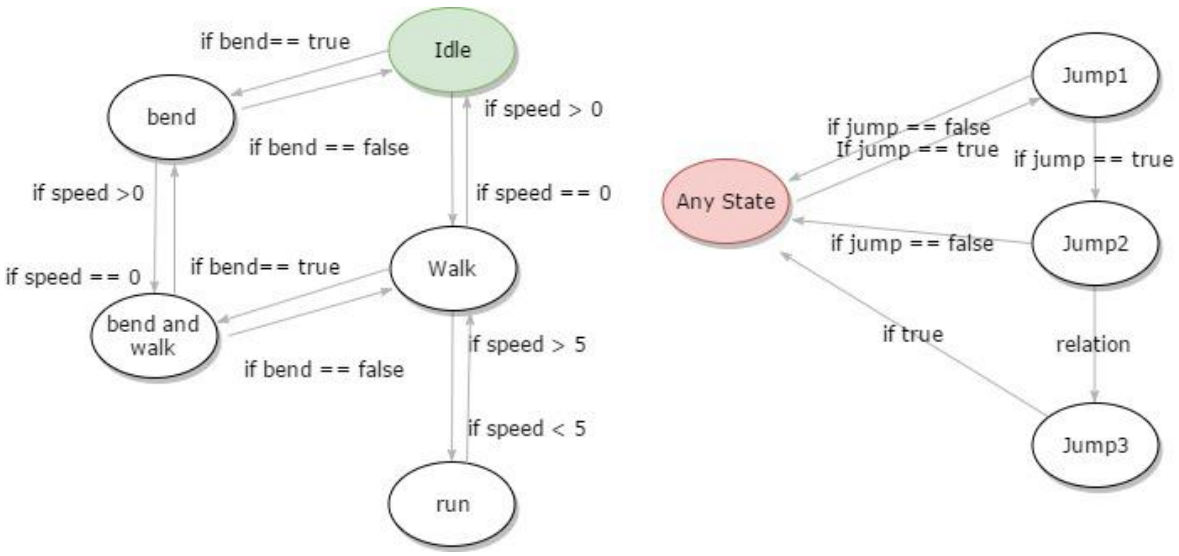


Figura 3- 2: Ejemplo de ActionMachine

La clase *State* contiene los datos necesarios para gestionar un estado. Cada estado esta formado por un blendtree (una lista de animaciones y pesos). El caso más extendido

será el que contiene una sola animación con un peso constante máximo. Cada estado se podrá configurar de forma adecuada a las necesidades del juego, permitiendo modificar la lista de pesos en tiempo real. A su vez cada estado contiene una o varias transciciones.

Una *Transition* consiste, básicamente, en una función condición de activación que devuelve un valor *true* o *false* y depende de los parámetros o propiedades de la ActionMachine , un estado objetivo y una duración. La función condición es la encargada de verificar si se dan las condiciones adecuadas para que se active dicha transición. Una vez se activa la transción se deberá programar el *fadein* y el *fadeout* para que realice el cambio gradual de un estado a otro en el tiempo especificado por la duración. Si este tiempo es corto será una transición más brusca.

La figura 3-2 muestra un ejemplo práctico de un ActionMachine. En el encontramos los estados: idle, walk, run, bend, bend and walk, jump1, jump2 y jump3. Por otro lado disponemos de las variables de control speed, bend y jump. A medida que vayamos avanzando en el desarrollo del sistema podremos profundizar más en detalle sobre este ejemplo.

3.1.3 ActionMachine

Como ya hemos dicho este es el corazón del motor. Esta clase es hija de la clase *BlendCharacter* aportada por three.js, por tanto las funciones usadas para cargar el modelo son las mismas y es posible añadir directamente tu *ActionMachine* a la escena THREE.js con la que se esté trabajando El principal motivo por el que he decidido que esta clase herede de *BlendCharacter* es para simplificar la integración dado que, al final, este proyecto es una pieza más a añadir sobre un proyecto en three.js.

La clase debe permitir en primera instancia funciones que proporcionen una interfaz y faciliten la implementación de la máquina de estados. Estas serían:

- **setProperty(nombre, valor) y getProperty(nombre):** Estas funciones proporcionan la interfaz para alterar el estado de la máquina a lo largo del videojuego, así como para inicializar las propiedades o parámetros de la máquina.

- **addState(state) y getState(name):** Interfaces que permiten insertar y recuperar estados en la máquina.

3.1.4 State

Para aumentar la funcionalidad del motor cada state se relaciona con un conjunto de animaciones y pesos o, lo que es lo mismo, un *blendtree*. Esto permite asignar a cada state una funcionalidad más compleja sobrescribiendo la función *updateWeights(params)* que se encargaría de insertar a cada animación el peso concreto en función de los parámetros (de forma similar a la vista en el apartado 2.2.3).

En cada máquina de estados existen dos estados especiales:

- **iniState:** Es un estado normal al que se le asigna la labor de ser el estado inicial o el estado por el que comienza a ejecutarse la máquina.

- **anyState:** Este es un estado virtual que no tiene ningúna acción asignada, solo transiciones que pueden contenerle como origen o destino. Las transiciones que le contienen como origen son transiciones que se deben verificar siempre pues significa que se pueden activar en cualquier momento. Cuando se activa una de estas transiciones se guarda el estado donde se encontraba la máquina y se ejecuta la nueva transición. Cuando se activa una transición que tiene por objetivo el estado virtual anyState se ejecuta el estado guardado en la memoria (o el inicial si no hay ninguno).

3.1.5 Diagrama de flujo

El siguiente diagrama de flujo representa la secuencia update que realiza la ActionMachine en cada ciclo del juego. He decido representar esta funcionalidad porque es la más profunda, y la que en definitiva se ejecutará.

En definitiva lo que tiene que hacer la máquina cada ciclo es ejecutar la animación actual y verificar si se da alguna de las condiciones de cambio; si se da programa la transición y cambia el estado actual.

Hay comprobaciones que, por simplificar, he decidido dejar fuera del diagrama, como las comprobaciones especiales de anyState o las que verifican si se ha terminado la animación actual y si hay que volver a ejecutarla.

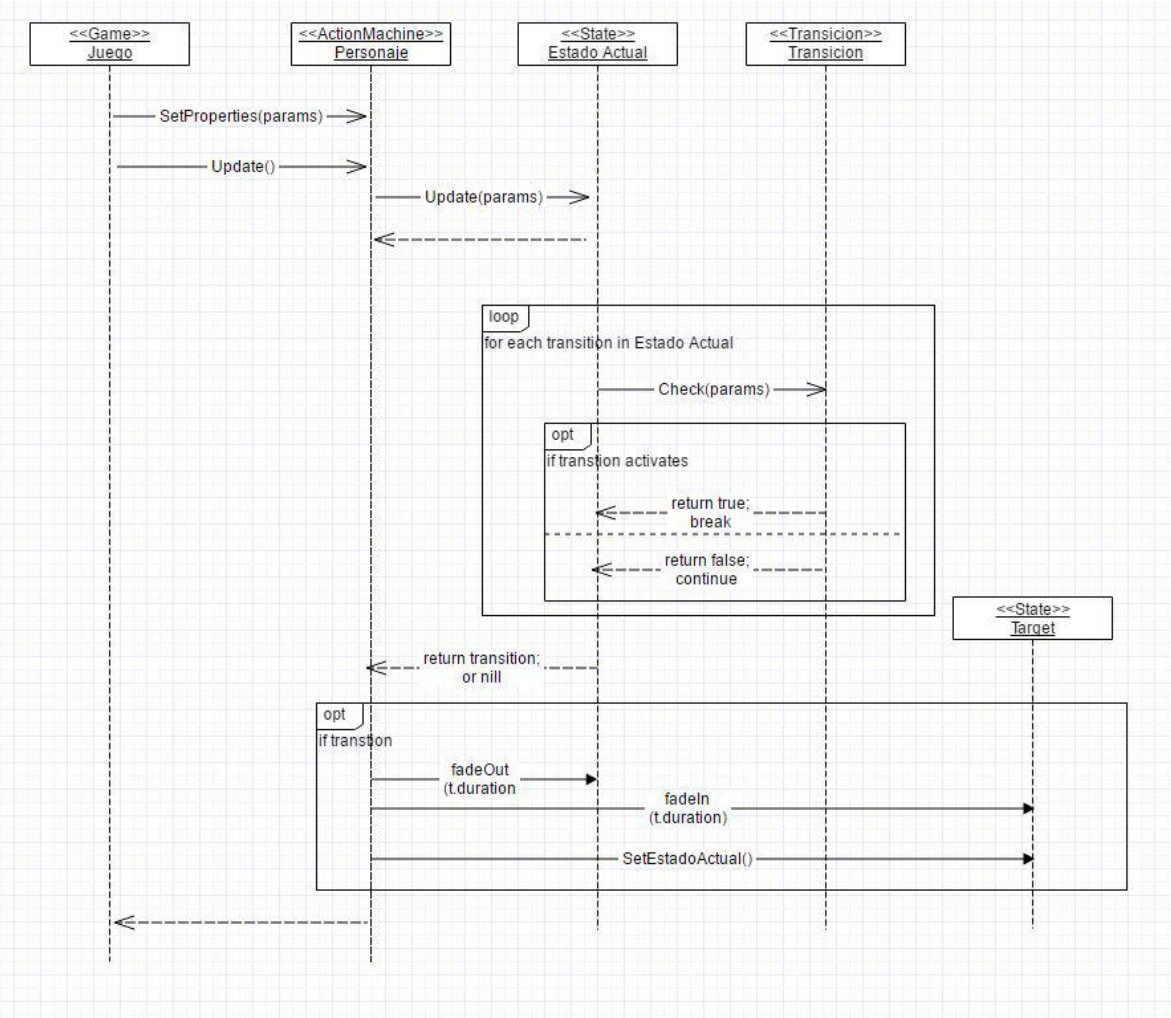


Figura 3- 3: Diagrama de flujo

3.2 El editor de ActionMachine

3.2.1 Analisis de requisitos

El editor debe ser una herramienta cómoda y sencilla que permita editar y visualizar de forma gráfica una ActionMachine, relacionada con un SkinnedMesh concreto.

Nº	Requisito
1	Permitir cargar y visualizar el modelo sobre el que se construye la máquina de acciones
2	Identificar correctamente las posibles animaciones del modelo y enseñarselas al usuario
3	Insertar, modificar y eliminar de forma gráfica estados y transiciones.
4	Poder mover los estados y la cámara del editor de forma cómoda e intuitiva
5	Interactividad con el ratón
6	Editar los parámetros de cada estado y transición

7	Insertar, modificar y eliminar propiedades y valores a la maquina
8	Poder descargar un fichero que guarde la información de la máquina de estados para poder guardar el progreso
9	Cargar el fichero descargable correctamente para continuar con la edición de la máquina de estados
10	Generar de forma correcta el código javascript correspondiente

3.2.2 Diseño

El editor a un nivel puramente lógico es sencillo y está dividido en cuatro módulos principales:

- **Interfaz y control de botones:** Esta es la sección encargada de actualizar y mostrar por pantalla los formularios necesarios y asegurarse de coordinar los datos introducidos por el usuario con la estructura interna de datos.

- **Editor gráfico de la máquina de estados:** Este módulo es el principal y más importante de diseñar bien. Corresponde a todas las clases y la lógica necesarias para pintar e interactuar con el canvas a modo de editor.

- **Mesh loader, previewer y poblado de las acciones:** Módulo encargado de integrar una escena three.js, cargar el modelo, mostrarlo en la ventana de previsualización y sacar las animaciones asociadas al modelo para mostrarselas al usuario.

- **Guardado, cargado y exportación de código:** Esta es la parte más esencial de la funcionalidad, ya que la idea de el editor es poder generar el código javascript de forma correcta para instanciarlo en la aplicación o videojuego que se esté desarrollando.

Estos tres módulos funcionan de forma conjunta para proporcionar la experiencia de usuario completa.

En la figura 3-4 se muestra la maqueta de la interfaz.

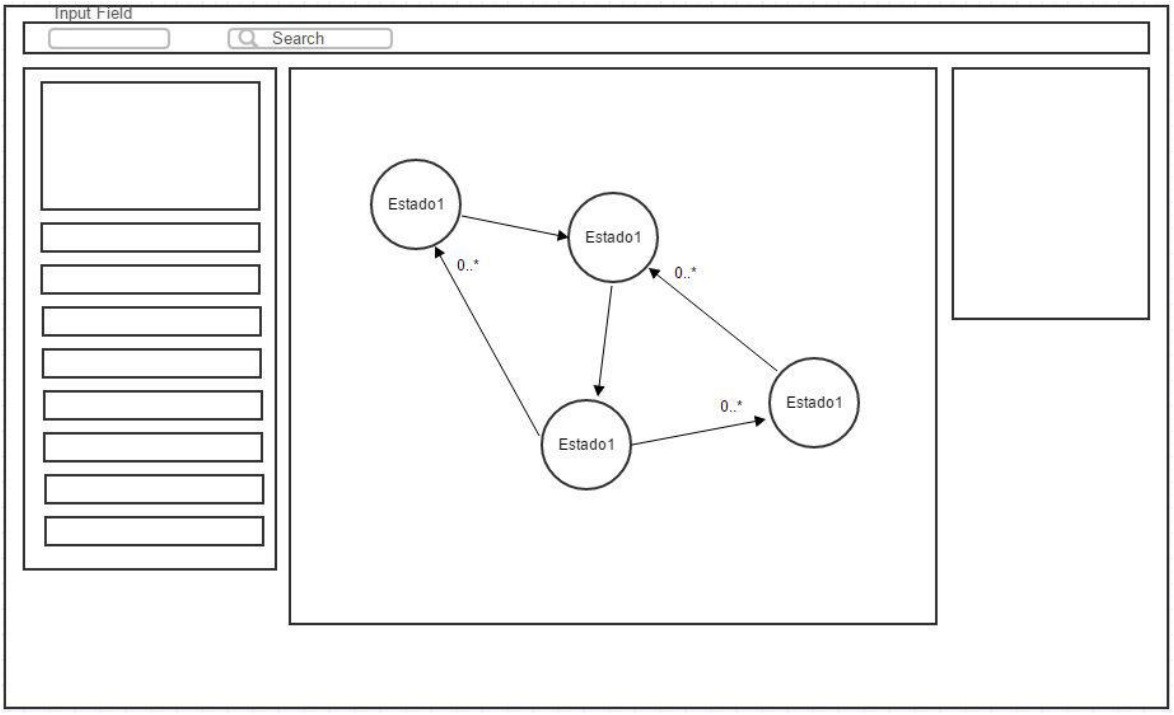


Figura 3- 4: Maqueta de la interfaz

3.2.3 Editor gráfico de la máquina de estados

El editor gráfico es la parte más compleja de diseñar y programar. Para crear una interfaz cómoda y usable he definido los siguientes requisitos:

- **Poder moverse por el lienzo con el ratón:** Esto es permitir que haciendo click en cualquier parte del lienzo se pueda arrastrar este. También permitir usar la rueda para hacer zoom-in y zoom-out.

- **Poder crear, mover, seleccionar y eliminar estados con el ratón y atajos del teclado:** esto implica gestionar de forma adecuada multitud de eventos para capturar las acciones del jugador e implementar un sistema de drag&drop para los estados

- **Crear transiciones que unan estados, seleccionarlas y eliminarlas:** Las transiciones, que se representarían gráficamente como flechas

Para lograr esta funcionalidad se usarán los siguientes elementos:

- **Cámara:** La cámara es un concepto que permite aplicar una transformación geométrica a todos los elementos antes de pintarlos de tal forma que permita moverse adecuada y cómodamente por la máquina de estados. Tiene tres propiedades principales; las coordenadas x e y y la escala. Las dos funciones más importantes son las que permiten transformar vectores de el sistema de coordenadas real al que se usa para pintar a la pantalla aplicando la fórmula:

```
drawX =  (realX - CameraX) * CameraScale
```

De esta forma basta con modificar los valores de CameraX, CameraY y CameraScale para moverse por el entorno.

- **Estados:** Poseen unas coordenadas x e y que se guardan en coordenadas reales (y siempre que se quiera pintar habrá que filtrarlas con la cámara). Tienen una función de dibujo que pinta el estado de forma correcta en la pantalla, aplicandole los efectos que sean necesarios (por ejemplo, si está seleccionado remarcando el estado). También, dada la posición del ratón, deben saber ver si están siendo pinchadas por el mismo o no.

También almacenan toda la información relevante de los estados.

- **Transiciones:** Tienen un estado origen y final, al igual que los estados tienen una función de dibujo y otra que permite saber si están siendo seleccionadas por el ratón. Las transiciones duplicadas se eliminarán automáticamente.

En la figura 3-5 se muestra un diagrama de flujo de la función repaint() que ayuda a entender la interacción entre estos componentes. Esta función es la encargada de limpiar la pantalla y pintar todos los elementos y hay que llamarla siempre que haya un cambio.

La última cuestión importante a destacar es la captura y respuesta a los diferentes eventos que garantizan una buena interactividad. En la siguiente tabla se resumen los eventos gestionados:

Evento	Comprobaciones	Respuesta
mouseDown	Si el ratón se encuentra sobre un estado	Prepara el estado para moverlo con el ratón. Activa listeners mouseMove y mouseUp
	Si no	Prepara la cámara para moverla con el ratón. Activa listeners mouseMove y mouseUp
	Si el ratón se encuentra sobre un estado y esta activada la flag SHIFT	Crea una transición con origen en el estado pulsado. Activa listeners mouseMove y mouseUp
onDblClick	Si el ratón se encuentra sobre un estado	Selecciona el estado
	Si el ratón se encuentra sobre una transición	Selecciona la transición
	Si está activada la flag SHIFT y no se encuentra en ningún estado / transición	Crea un estado nuevo
mouseMove		Desplaza la cámara / el estado seleccionado / el punto final de la transición recién creada
onkeyDown	Si la tecla pulsada es SHIFT	Activa la flag SHIFT
	Si la tecla pulsada es DELETE	Elimina el estado / transición seleccionada (si hay)

onKeyUp			Desactiva la flag SHIFT
mouseUp			Finaliza el movimiento tipo drag de la cámara o el estado seleccionado
	Si se está creando una nueva transición	Si el ratón está sobre un estado existente	Asigna como target de la nueva transición el estado sobre el que se encuentra el ratón
		Si no	Crea un nuevo estado que será el target de la transición creada en la posición del ratón
MouseWheel	Si la rueda se mueve en dirección +1		Aumentar la escala de la camara
	Si la rueda se mueve en dirección -1		Reducir la escala de la cámara

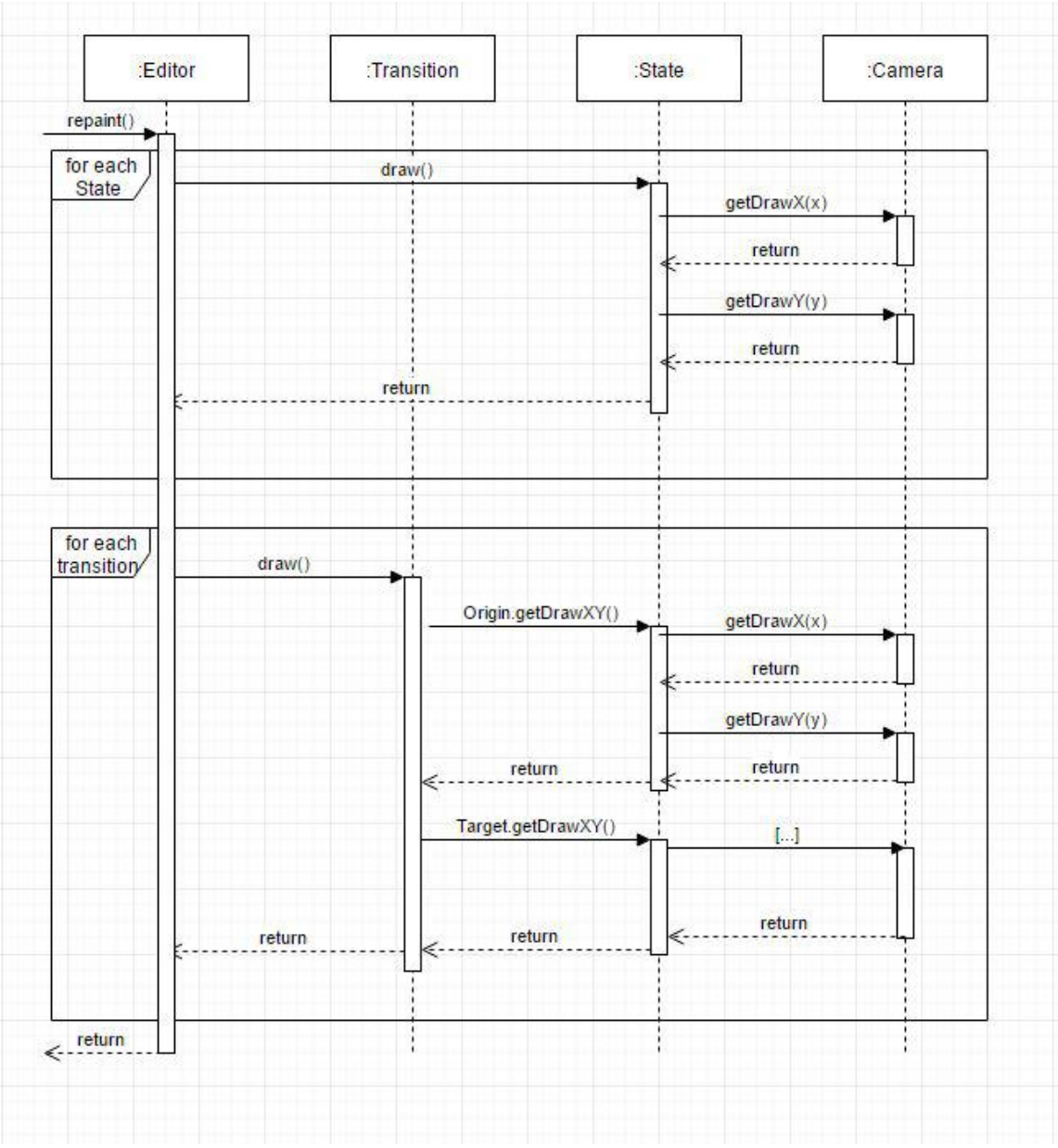


Figura 3- 5: Flujo de la función repaint

4 Desarrollo

4.1 Tipo de desarrollo

Para la elaboración de este proyecto se ha usado un desarrollo de tipo incremental [9], planificando tres hitos u objetivos principales, divididos en las cuatro tareas principales de: investigación/documentación, análisis y diseño, desarrollo y pruebas. Estos hitos se han distribuido de forma homogénea entre las 8 semanas de producción.

Dado que es un proyecto elaborado por un solo desarrollador no ha habido ninguna dificultad a la hora de aplicar cambios de diseño o durante las fases de desarrollo pudiendo seguir un proceso flexible. En cualquier caso no he tenido que realizar ningún cambio importante, tan solo pequeños añadidos que no se tuvieron en cuenta en las fases iniciales.

4.2 Principales hitos y planificación del proyecto

Este proyecto se ha realizado a lo largo de los meses de Abril y mayo siguiendo la aquí expuesta planificación:

Semana del 3 de Abril	Preparación del entorno de desarrollo. Búsqueda de bibliografía, documentación y referencias
Semana del 10 de Abril	Estudio de documentación sobre WebGL y Three.js, estudio de la API y lectura del código
Semana del 17 de Abril	Analisis, diseño e implementación del motor ThreeAction
Semana del 24 de Abril	Previsualizador de ActionMachines y pruebas
Semana del 1 de Mayo	Lectura y búsqueda de tecnologías para desarrollar el editor
Semana del 8 de Mayo	Estudio de HTML5 Canvas. Diseño del editor
Semana del 15 de Mayo	Desarrollo del Editor y pruebas
Semana del 22 de Mayo	Pruebas y finiquitar la memoria

4.2.1 Principales hitos

Los principales hitos han sido:

- **Elaboración y pruebas del motor de animación**; para el 30 de Abril. Se deberá tener la estructura del motor finalizada y preparada para ser implementada en juegos.

- **Elaboración y pruebas del editor de ActionMachines**; para el 21 de Mayo. Se deberá tener el editor acabado y la metodología o workflow para poder intergrarlo en el juego.

- **Memoria del proyecto**; para el 29 de Mayo. Esta se ha ido elaborando de forma paralela al desarrollo del proyecto.

4.3 Estructura del proyecto

En esta sección explicaré los archivos y carpetas principales que componen el proyecto.

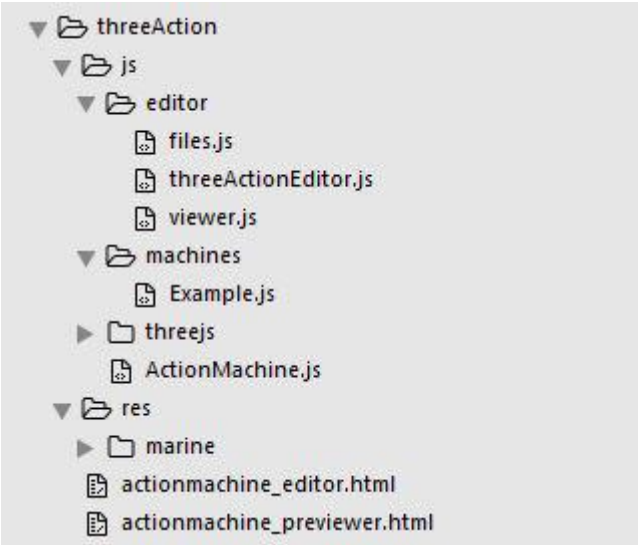


Figura 4- 1: Estructura del proyecto

- **Actionmachine_editor.html:** Este fichero es el editor de máqunas de estado
- **Actionmachine_previewer.html:** Previsualizador de máquinas de estado para pruebas
- **Js/editor:** Guarda todo el cñodigo js necesario para el editor y que se ha extraido por módulos del fichero html.
  - threeActionEditor.js: Incluye todo el código relacionado con el editor gráfico
  - files.js: Módulo de guardado y cargado. Los ficheros que se guardan se descargan del navegador y los que se cargan se suben desde el terminal local.
  - viewer.js: Código del previsualizador del modelo y de cargado del mismo.
- **Js/ActionMachine.js:** Código del motor y clases State, Transition y ActionMachine.
- **Js/Three.js:** Incluye todos los ficheros de la librería three.js que son necesarios.
- **Js/machines:** Código js de las máquinas de estado
- **Res**: Recursos gráficos (skinned mesh en formato .json y texturas)

4.4 Integración con three.js

Para integrar correctamente los ActionMachine en un entorno o videojuego creado con three.js he tenido que enfrentarme a la decisión inicial de somo integrar la clase ActionMachine con la clase SkinnedMesh, dado que, ActionMachine es, en el fondo, un gestor de un objeto del segundo tipo. Principalmente disponía de 2 opciones:

- Integrar un SkinnedMesh como un componente de la clase ActionMachine.

- Hacer que la clase ActionMachine herede y expanda esta clase.

He considerado esta segunda opción como la más facilitadora a la hora de integrar un ActionMachine como un elemento más de la escena. Otro motivo por el que he decidido usar la herencia como solución es porque me permite integrar el elemento de forma más directa al heredar de la clase object de three.js que es la que representa los elementos que se pueden incluir en una escena.

Un programador que haga uso de este motor no tendrá que preocuparse más que por familiarizarse con dos o tres funciones para instanciar e interactuar con la máquina. Por lo demás su uso es completamente analogo al de cualquier objeto de three.js, y el motor gestionará de forma automática las animaciones y las transiciones entre ellas de forma transparente al programador.

Para integrar correctamente el motor con three.js había varios aspectos a tener en cuenta:

- **Cargado del modelo / load:** Para ello me he inspirado en la función incluida en la clase BlendCharacter que se encuentra en los archivos de ejemplo de Three.js.

- **Gestión de las animaciones:** No solo es la gestión de los estados y las transiciónes, sino también asegurarse de que las animaciones se van actualizando a cada ciclo.

- **Integración en el entorno / la escena:** Dado que las ActionMachines heredan directamente de SkinnedMesh, pasan a ser un elemento más de Three.js que se deriva de la clase Object la cual contiene las funciones principales que permiten añadirle a una escena, mover y rotarle de la misma forma que se haría con un cubo o cualquier otro elemento del motor.

4.4.1 La función load

La función load ha sido tomada de BlendCharacter.js. Es una de las más complejas porque se encarga de hacer varias cosas a un nivel muy bajo.

De forma general es una llamada a la función load del objeto loader de Three.js pasandole la URL del elemento que se quiere sacar. La principal dificultad se encuentra en la función onLoad (función que se ejecuta cuando el archivo a terminado de leerse).

Dado que no se permite exportar SkinnedMesh hay que rescatarlo manualmente recorriendo la jerarquía para lo que se sobescribe la función traverse (que es una función

tipo callback, que se se ejecuta sobre todos los hijos del objeto). Después hay que definir el skinnedMesh con sus materiales.

En última instancia es necesario crear el animationMixer, que es el gestor encargado de contener y ejecutar los clipActions.

Aquí dejo el código de la función load:

```
this.load = function ( url, onLoad ) {

    var scope = this;

    var loader = new THREE.ObjectLoader();
    loader.load( url, function( loadedObject ) {

        // The exporter does not currently allow exporting a skinned mesh by itself
        // so we must fish it out of the hierarchy it is embedded in (scene)
        loadedObject.traverse( function( object ) {

            if ( object instanceof THREE.SkinnedMesh ) {

                scope.skinnedMesh = object;

            }

        } );

        THREE.SkinnedMesh.call( scope, scope.skinnedMesh.geometry, scope.skinnedMesh.material );

        // If we didn't successfully find the mesh, bail out
        if ( scope.skinnedMesh == undefined ) {

            console.log( 'unable to find skinned mesh in ' + url );
            return;

        }

        scope.material.skinning = true;

        scope.mixer = new THREE.AnimationMixer( scope );

        // Create the animations
        for ( var i = 0; i < scope.geometry.animations.length; ++ i ) {

            scope.mixer.clipAction( scope.geometry.animations[ i ] );

        }

        // Loading is complete, fire the callback
        if ( onLoad !== undefined ) onLoad();

    } );

};
```

Figura 4- 2: Función load

4.4.2 Uso del AnimationMixer de SkinnedMesh

Para gestionar las animaciones he tenido que utilizar de forma directa la clase mixer. Para entenderlo, la clase AnimationMixer es un reproductor que gestiona los AnimationClips de un objeto concreto y se encarga de reproducirlos.

Un animationClip es un conjuto de keyframes que representan una animación. Un keyframe es una posición concreta y estática del elemento situada en un momento determinado que, en conjunto con otras keyframes e interpolando el tiempo entre ellas define una animación.

El AnimationMixer es la clase con la que más trabaja la clase ActionMachine y State teniendo que:

- Anunciarle que **animaciones se están ejecutando** en cada momento y **con que pesos.**
- Programar los **fadeIn** y **fadeOut** de las animaciones con sus respectivos pesos
- Llamar a la función **mixer.update** a cada ciclo de reloj para que se ejecuten las animaciones correctamente.

4.5 Desarrollo del editor

4.5.1 Por que en HTML5

Desde el inicio he tenido bastante claro que quería que el editor de ActionMahines estuviese desarrollado en HTML5. Dado que es el entorno hacia el que todo el proyecto está orientado, lo más coherente es aprovechar las tecnologías que se estan potenciando en este proyecto. Por otro lado también me parece la opción más cómoda de cara al desarrollador que esta construyendo su proyecto sobre HTML5 dado que tendrá más facilidad para modificar el editor a su gusto, añadiendo o cambiando lo que viese necesario para sus necesidades concretas.

Además, este entorno no requiere instalación alguna y, tal como ha sido implementado el editor, se puede trabajar en un entorno local estando el editor alojado en un servidor.

4.5.2 Posibles tecnologías para el editor gráfico

Hay cientos de librerias que facilitan el uso de canvas y de interfaces y GUIs en general de HTML5. Algunas de las que he estado estudiando son Zebra.js, Go.js o SVG. La principal ventaja de usar este tipo de librerías es que abstraen el uso de canvas simplficicando ciertas tareas que para un programador inexperto en cuestiones de gráficos pueden ser difíciles de interiorizar.

Pero después de hacer un estudio exahustivo de este tipo de librerías y viendo el funcionamiento de canvas vi que el tiempo que tardaría en aprender a manejar y adaptar a mis necesidades una de estas librerías era similar al que tardaría en desarrollar directamente una aplicación sobre el canvas dado que ya estoy familiarizado con la programación gráfica y la API de canvas sobre javascript no es especialmente compleja.

Por ello tras hacer una valoración suficiente opte por utilizar directamente canvas sin una API intermedia.

4.5.3 Aspectos técnicos

4.5.3.1 La función draw de state

Cada elemento debe contener una función draw que se encarga de dibujar el objeto en el canvas siempre que haga falta. Esta función es en la que se hace un uso directo de la API de canvas mientras que, las demás, tan solo actualizan los valores de los elementos. Los estados son relativamente sencillos de dibujar, tan solo hay que tener en cuenta las transformaciones de la camara a la hora de definir el tamaño y la posición en la que se tiene que dibujar el elemento o el tamaño del texto.

Hay que tener en cuenta las propiedades del estado (si está o no seleccionado; si es o no iniState, o anyState) para definir que parametros utilizar.

La API de canvas permite dibujar aplicando comandos al objeto context (que pertenece al elemento HTML5 canvas). Es un proceso secuencial en el que defines una forma, después defines el relleno y la línea de la forma.

```
// Función que dibuja el estado dónde debe
this.draw = function(camera, context){
    // Dibujamos el círculo
    context.beginPath();
    context.arc(camera.getDrawX(this.x),
        camera.getDrawY(this.y),
        camera.getDrawSize(50), 0, 2 * Math.PI);

    if (this.iniState){
        context.fillStyle = '#9bae59'
    }else if (this.anyState){
        context.fillStyle = '#b5875b';
    }else{
        context.fillStyle = '#282828';
    }

    if (this.selected){
        context.strokeStyle = '#ffe325';
        context.lineWidth = 5;
    }

    context.fill();
    context.stroke();
    context.strokeStyle = 'black'; context.lineWidth = 1;

    // Dibujamos el texto
    context.font = "bold " + Math.floor(camera.getDrawSize(25)) + "px Arial";
    context.fillStyle = '#7c7c7c';
    context.textAlign = "center";
    context.fillText(this.name, camera.getDrawX(this.x),camera.getDrawY(this.y));
    context.strokeText(this.name, camera.getDrawX(this.x),camera.getDrawY(this.y));
}
```

Figura 4- 3: Función draw de State

4.5.3.2 La función draw de transition

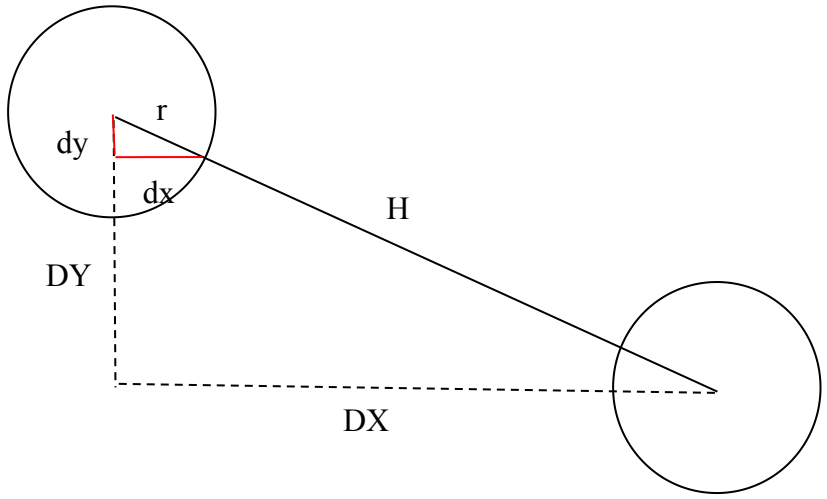
La función draw de transition tiene mucha más complicación. Para empezar existen dos modos de visualización de las transiciones y, por tanto, dos funciones distintas para dibujarlas:

- **Draw_creating:** Dibuja una línea entre el estado origen y el ratón (caundo aún no está definido el target y se está realizando la acción de drag&drop)

- **Draw:** Función estandar que se utiliza en la mayoría de los casos que dibuja la flecha entre dos estados.

Por otro lado no existe una función por defecto en canvas que sirva para dibujar flechas, por tanto he hecho uso de la función drawArrow que he sacado de la dirección: http://stackoverflow.com/questions/808826/draw-arrow-on-canvas-tag con alguna modificación.

La función drawArrow necesita un punto origen y un punto destino para dibujar la flecha, y los estados guardan el centro del estado solamente. Eso significa que la función **draw** tiene que calcular el punto de corte entre la línea que une los centros de los estados y la circunferencia que lo representa.



Es un cálculo de trigonometría muy sencillo, solo hay que calcular dx y dy, conociendo H, DX, DY y r es muy sencillo aplicando el Teorema de Thales [10].

Este cálculo se puede ver en la función draw mostrada en la figura 4-4

```
this.draw = function(camera, context){
    if (this.target == -1){
        this.draw_creating(camera, context);
        return;
    }

    if (this.selected){
        this.color = "#ffe325";
    }else{
        this.color = 'black';
    }

    // Pasamos todo a coordenadas de dibujo
    fromx = camera.getDrawX(this.oOrigin.x);
    fromy = camera.getDrawY(this.oOrigin.y);
    tox = camera.getDrawX(this.oTarget.x);
    toy = camera.getDrawY(this.oTarget.y);

    // Calculamos los puntos inicial y final de la línea
    // No queremos que sea el centro de los círculos, sino
    // un punto en el círculo
    // para que se aprecie la flecha
    DX = tox - fromx;
    DY = toy - fromy;
    H = Math.sqrt(DX*DX + DY*DY);

    dx = DX * camera.getDrawSize(50) / H
    dy = DY * camera.getDrawSize(50) / H

    fromx = fromx + dx;
    fromy = fromy + dy;
    tox = tox - dx;
    toy = toy - dy;

    this.arrowX = tox;
    this.arrowY = toy;

    this.drawArrow(fromx, fromy, tox, toy, context);
    context.fillStyle = 'black';
    context.lineWidth = 1;

}
```

Figura 4- 4: Función draw de transition

4.5.3.3 La generación de código

Uno de los aspectos más importantes es la generación de código dado que, en definitiva, es el fin último del editor. Esta generación se realiza de forma recursiva accediendo a cada uno de los elementos (states y transitions) que tienen una función interna toCode. Esta función retorna un string que contiene el código necesario para definir ese estado o esa transición en la máquina de estados.

El código generado está dividido en 5 bloques:

- **Cabecera:** Se inicializa la función _NombreDeLaMaquina_() que devuelve una instancia de la máquina que se quiere crear. Se instancia la variable machine como un nuevo elemento de la clase ActionMachine.

- **Propiedades:** Se insertan cada una de las propiedades con sus valores por defecto utilizando la función machine.setProperty().

- **Estados**: Se insertan los estados haciendo uso de la función machine.addState() pasandole un identificador, el array de acciones y el array de pesos.

- **Transiciones:** Se insertan las transiciones. Las transiciones deben insertarse no en la máquina de estados sino en el estado de origen. Por ello, para insertar las

transiciones se recuperan los estados haciendo uso de la función machine.getState() y se insertan con la función state.addTransition()

- **Final:** Al final se carga el modelo llamando a la función load, se devuelve la máquina y se cierra la función.

En la siguiente figura se puede ver un ejemplo final de código generado:

```
function _Marine_(){

    var machine = new THREEACTION.ActionMachine();

    machine.setProperty(tpose, false);
    machine.setProperty(speed, 0);


    machine.addState(new State('idle', ['idle'], [1]));
    machine.initState = getState('idle');
    machine.addState(new State('walk', ['walk'], [1]));
    machine.addState(new State('run', ['run'], [1]));
    machine.addState(new State('tpose', ['Tpose'], [1]));


    machine.getState('idle').addTransition(new Transition(machine.getState('walk'), 0.5, function(params){
    return params.speed > 0;
}));
    machine.getState('walk').addTransition(new Transition(machine.getState('run'), 0.5, function(params){
    return params.speed > 5;
}));
    machine.getState('run').addTransition(new Transition(machine.getState('walk'), 0.5, function(params){
    return params.speed < 5;
}));
    machine.getState('walk').addTransition(new Transition(machine.getState('idle'), 0.5, function(params){
    return params.speed == 0;
}));
    machine.getState('AnyState').addTransition(new Transition(machine.getState('tpose'), 0.5, function(params){
    return params.tpose;
}));
    machine.getState('tpose').addTransition(new Transition(machine.getState('AnyState'), 0.5, function(params){
    return !params.tpose;
}));


    machine.load( 'res/marine/marine_anims_core.json', start );
    return machine;
}

```

Figura 4- 5: Código generado

4.5.4 Resultado final

En las figuras 4-2 muestro el resultado final del editor de forma general.

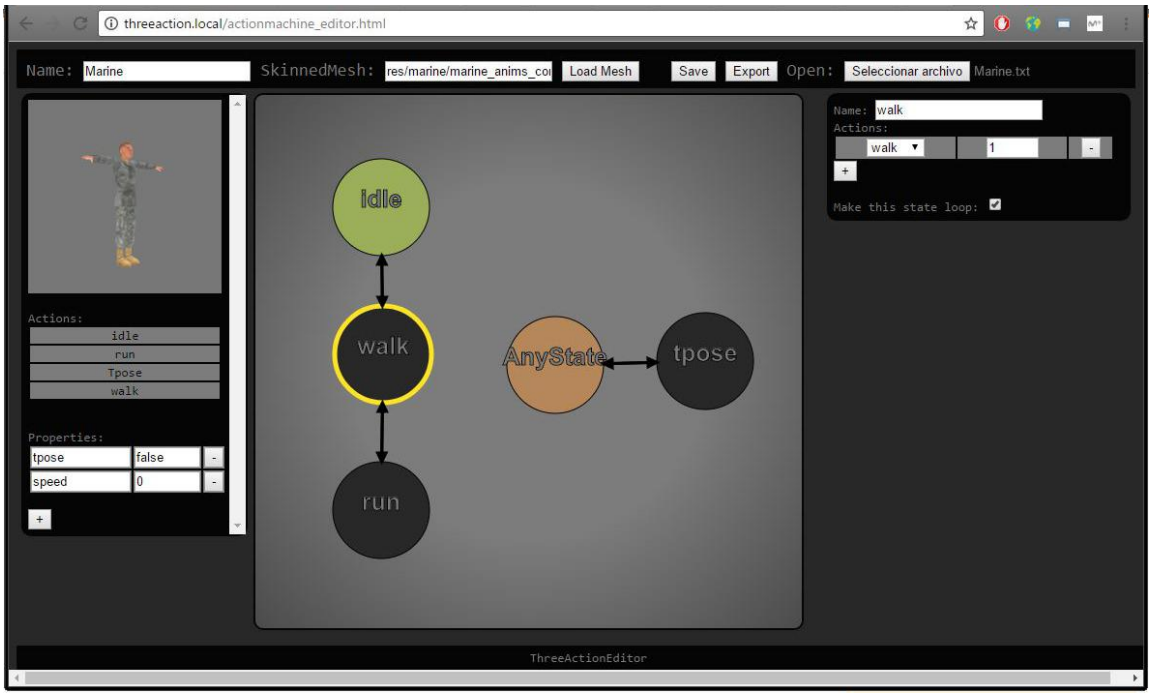


Figura 4- 5: Resultado final de la interfaz

En el Anexo B hay una explicación detallada de cada una de las partes del editor.

4.6 Apache y WAMP como entorno de desarrollo

Existen grandes limitaciones en javascript a la hora de escribir y leer ficheros en el entorno de usuario por motivos de seguridad evidentes. Este problema se soluciona ejectuando el código desde un servidor donde se alojen los modelos y elementos que el programa pueda necesitar cargar.

Para el desarrollo de este proyecto he utilizado Apache 2.4.2 y WAMPSERVER 2.2 como entorno de pruebas. Wamp ofrece muchas prestaciones (PHP y MySQL) que no son necesarias en este proyecto, pero por facilidad he decidido utilizar este entorno.

5 Integración, pruebas y resultados

5.1 Pruebas y resultados del motor ThreeAction

Para realizar las pruebas en el motor ThreeAction se ha implementado la aplicación **Actionmachine_previewer.html** que permite visualizar el modelo y las animaciones en tiempo real, a la vez que se editan las propiedades del objeto. Se ha utilizado el modelo marine_animation_core.json que se incluye en el source de three.js para realizar las pruebas.

Los casos de prueba realizados han sido las siguientes:

- Comprobar que las transiciones entre estados se ejecutan y programan correctamente y en el tiempo especificado por la transición.
- Verificar que las transiciones se activan de en el momento correcto.
- Ver que funciona el estado especial AnyState y se activan ls transiciones cuando deben, y ver que la acción de retorno al último estado ejecutado se cumple también.
- Comprobar que se recoge de forma correcta el evento *finished*
- Probar a insertar una función updateWeights a un estado y verificar que se comporta de forma correcta.

Con estas pruebas se verifica un funcionamiento correcto del motor en todas las posibilidades dadas. Se ha realizado una prueba individual para cada uno de estos casos y una prueba general que recoge todas estas opciones.

Los resultados han sido satisfactorios y verifican el correcto funcionamiento de la clase ActionMachine.

5.2 Pruebas y resultados del editor de ActionMachine

5.2.1 Pruebas de funcionalidad

Hay que verificar que el editor responde de forma correcta a la siguiente batería de pruebas:

- Verificar que las acciones básicas del editor de drag&drop, crear y eliminar funcionan de forma correcta y sin lags ni errores. También comprobar que la cámara se mueve.

- Comprobar que la selección funciona correctamente, se destaca el elemento seleccionado y se actualiza el pamel de la izquierda con los datos correctos.

- Verificar que se cargan de forma correcta las acciones del modelo usado y se actualizan en los formularios donde se tienen que actualizar.

- Comprobar que el guardado y el cargado de el estado del editor funciona de forma correcta y cubre todos los casos, especialmente máquinas de estados complejas que contengan múltiples acciones y propiedades.

- Verificar que la exportación de código se realiza de forma correcta para todos los casos.

5.2.2 Pruebas de usabilidad

Las pruebas de usabilidad han consistido en solicitar a un usario que desconozca la aplicación que haga un uso normal de ella tras una explicación sencilla y superficial. Después se han recogido las impresiones del usuario y se han sacado las siguientes conclusiones:

- El flujo de trabajo es cómodo y adecuado

- La interfaz funciona perfectamente y en poco tiempo se entiende dónde está cada cosa

- Se echan en falta funcionalidades estándar de otros editores como el comando CTRL-Z (*undo*), la multi selección, *copy*, *cut* y *paste*. La falta de este tipo de comandos a los que el usuario está muy acostumbrado hace que el uso, al menos inicialmente, se torne más incómodo o engorroso

5.3 Pruebas de integración

Las pruebas de la sección 5.1 han sido repetidas pero esta vez diseñando el ActionMachine con ayuda del editor en vez de codificándolo a mano y el resultado ha sido el mismo.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Este proyecto ha sido un buen acercamiento a un motor de animación que se pueda usar en un proyecto real. Estoy bastante contento con el resultado porque creo que ha salido un producto útil y me planteo liberarlo en la red o enviarlo al equipo de desarrollo de three.js para que le den uso si lo ven útil. Como expuse en la sección de motivación esta herramienta que acabo de desarrollar la podré usar yo mismo en proyectos futuros.

También ha sido la primera vez en la carrera que he dispuesto de la libertad de definir y pensar un proyecto de principio a fin, siendo esta la parte que más me apasiona de la ingeniería informática. Por mi personalidad creativa esta libertad hace que disfrute mucho más el trabajo y tenga un mayor rendimiento, frente a los proyectos más definidos o las prácticas que consisten en seguir un guión predefinido y que se me hacen más pesados y tediosos.

Por esto, aunque ha supuesto una cantidad de trabajo importante, lo he realizado con gusto y no se me ha hecho difícil ni cuesta arriba. Por supuesto ha habido dificultades y momentos de bloqueo como en el desarrollo de cualquier aplicación informática, pero con paciencia los he ido resolviendo.

6.2 Trabajo futuro

Si bien este proyecto cubre gran parte de las características que presenta cualquier motor comercial, aún hay añadidos que se podrían hacer para aumentar la funcionalidad:

- Morphing: Al inicio de la memoria hablamos de las técnicas de morphing y blending. Mientras se ha integrado de forma exhaustiva el concepto de blending tanto con las transiciones entre estados como los blendtrees internos de cada estado, el morphing se ha dejado totalmente de lado. Se podría estudiar la forma de añadir el morphing al sistema actual, ya sea con un sistema de capas y máscaras como en Mechanim o cualquier otra opción.

- Controladores de huesos: Unreal 4 ofrece un sistema muy completo que permite añadir funcionalidad concreta a huesos específicos, haciendo que un personaje mire siempre a un lugar determinado o que la posición de los pies se adapte de forma correcta a la forma de una escalera o de suelos irregulares. Esta añadido supondría todo un reto porque muy probablemente implicaría estudiar y trabajar a un nivel mucho más profundo de Three.js y WebGL.

- Por otro lado ya se ha visto en la parte de pruebas de usabilidad (sección 5.2.2) del editor que la usabilidad aún es mejorable. Una posible mejora sería perfilar y limar estos detalles que harían que el editor fuera mucho más cómodo.

- Por otro lado el editor actualmente no cubre la edición de los blend trees internos de cada estado. Se podría expandir su funcionalidad para cubra tambiñen este caso de uso.

- Las condiciones en las transiciones hay que programarlas a mano, si bien el editor ofrece un espacio donde insertar el código, se podría buscar la forma de hacerlo tal que quien esté haciendo uso del editor no deba preocuparse por escribir código javascript.

- Si bien existe un parametro que permite capturar el evento de finalización de una animación concreta, se podría facilitar el caso de uso añadiendo una transición especial.

Referencias

[1] E. Kuryanovich, S. Shalom, R. Goldenberg, M. Paumgarten, D. Straus, S. Lee-Delisle, G. Renaudeau, J. Wagner, J. Berknoff, B. Danchilla y R. Hawkes, "HTML5 Games Most Wanted" ANAYA multimedia, 2012

[2] «WebGL» [online] https://www.khronos.org/webgl/wiki/Getting_Started

[3] «Rigging» [online] https://www.lifewire.com/what-is-rigging-2095

[4] «Skinning» [online] https://en.wikipedia.org/wiki/Skeletal_animation

[5] «Blending - Game Development» [online] https://gamedev.stackexchange.com/questions/22402/animation-blending-basics

[6] «Morphing» [online] https://en.wikipedia.org/wiki/Morph_target_animation

[7] «Guinness» [online] https://twitter.com/gwr/status/477261516006100993

[8] «Unreal oficial documentation» [online] https://docs.unrealengine.com/latest/INT/Engine/Animation/Persona/index.html

[9] «Desarrollo incremental - proceso software» [online] https://procesosoftware.wikispaces.com/Modelo+Incremental

[10] «Teorema de Thales- Profesor en línea» [online] http://www.profesorenlinea.cl/geometria/Teorema_de_Tales.html

Glosario

API	Application Programming Interface
HTML5	HyperText Markup Language, versión 5
Videojuego	Aplicación interactiva
Mecánica	Conjunto de reglas que definen la jugabilidad
Animador	Profesional encargado del modelado y animación de un personaje
Game Engine	Conjunto de rutinas que hacen posible el diseño y ejecución de un videojuego
Motor gráfico	Software usado para dibujar gráficos
Plugging	Software que se puede integrar como una parte más pequeña de un software mayor
Multiplataforma	Propiedad de un programa o un lenguaje que se puede ejecutar en varias plataformas sin necesidad de recompilarse
Texturas	Imagen 2D que aporta información sobre cómo debe renderizarse un elemento 3D, como puede ser el color o texturas de profundidad
Renderizar	Proceso de generación de una imagen a partir de un modelo tridimensional
OpenGL	Open Graphics Library
Shaders	También conocido como sobreadores. Programa que describe el modo en el que se renderiza y reacciona a la luz u otros efectos un modelo.
GPU	Graphics Processing Unit
Granja de render	Conjunto de ordenadores o clusters que se reparten los cálculos de renderizado
Fotograma	Imagen estática que compone una película
Lightmap	Datos pregenerados en un proceso de lightmapping que contienen información sobre el brillo de una superficie
Capa	Elemento que permite organizar elementos dado cierto espacio
Máscara	Sección de un espacio que se verá afectado por cierta capa
Máquina de estados	Modelo de comportamiento de un sistema con entradas y salidas, en donde las salidas dependen no sólo de las señales de entradas actuales sino también de las anteriores.

Anexos

A Manual de instalación del editor

El editor ActionMachineEditor está pensado para ejecutarse desde un servidor. Para el usuario puede ser más cómodo ejecutarlo desde un servidor local. En este anexo se explicarán los pasos que habría que seguir para realizar la instalación.

A.1 Prerequisitos

Es necesario tener instalado Apache, WAMPSERVER o cualquier servidor que se pueda ejecutar sobre el sistema operativo en el que se esté trabajando.

- Apache se puede descargar desde: https://httpd.apache.org/download.cgi
- WAMPSERVER se puede descargar desde: http://www.wampserver.com/en/download-wampserver-32bits/

Aclaro que WAMPSERVER incluye apache junto con PHP y MySQL, las cuales son utilidades que para ejecutar este proyecto concreto no son necesarias.

A.2 Archivo de configuración de Apache

Se puede acceder y modificar la configuración de apache escribiendo en el archivo httpd.conf alojado en /MiDirecciónHastaApache/apache2.4.2/conf/httpd.conf.

Por defecto WAMPSERVER tiene habilitado un directorio www que ya está habilitado y configurado para alojar una página web, mientras que restringe, por motivos de seguridad, el acceso al resto de directiorios.

Para habilitar cualquier otro directorio debes añadir una etiqueta Directory al archivo de configuración como la siguiente:

```
<Directory "c:/wamp/www/">

    Options Indexes FollowSymLinks
    AllowOverride All
    Require local

</Directory>
```

Sustituyendo el directorio que quieres habilitar por ‘c:/wamp/www/’. El segundo paso para facilitar las cosas sería crear un host virtual, de forma que no sea necesario introducir el directorio completo a la página web cada vez que se quiera hacer uso de ella. Otra solución alternativa sería utilizar alias.

Para insertar un hist virtual se utiliza la directiva VirtualHost como en el ejemplo que sigue:

```
NameVirtualHost 127.0.0.1

<VirtualHost 127.0.0.1>
   DocumentRoot "C:\wamp\www\TFG\src\WebContent\threeAction"
   ServerName threeaction.local
</VirtualHost>
```

Esta es la solución que he usado en mi instalación personal. DocumentRoot representa el directorio principal del virtual host que se está creando mientras ServerName sería el nombre.

Hasta aquí los cambios que deberíamos realizar en la configuración de apache. Una vez hechos habría que reiniciar el servicio para que se aplicaran.

A.3 Archivo hosts

Si tan solo relizamos la configuración anterior y probamos a entrar en nuestro virtual host desde nuestro navegador tendremos un error porque no se podrá resolver la petición DNS (o se nos mostrará una página totalmente distinta a la que buscabamos). Para corregir esto deberemos modificar el archivo host de nuestro Sistema Operativo para indicarle que queremos que busque nuestra dirección en local.

Dependiendo del SO el archivo host estará en un lugar u otro (puede cambiar incluso de unas versiones de Windows a otras). Deberás localizarlo siguiendo las intrucciones específicas del mismo y, una vez hecho, abrirlo con algún editor de texto que puedas utilizar con privilegios de administrador.

En el archivo tendrás que añadir una única línea que sería como sigue:

```
127.0.0.1    threeaction.local
```

Tras realizar estos pasos ya debería estar todo listo para comenzar a usar el editor.

B Guía de uso de ActionMachineEditor

Este anexo está destinado a proporcionar una guía completa para aquella persona que vaya a usar el editor para diseñar el comportamiento de sus personajes en un videjuego, ya sea artísta, programador o artista técnico. En él se explicarán las partes que componen el editor, para que sirve cada una y como se pueden utilizar.

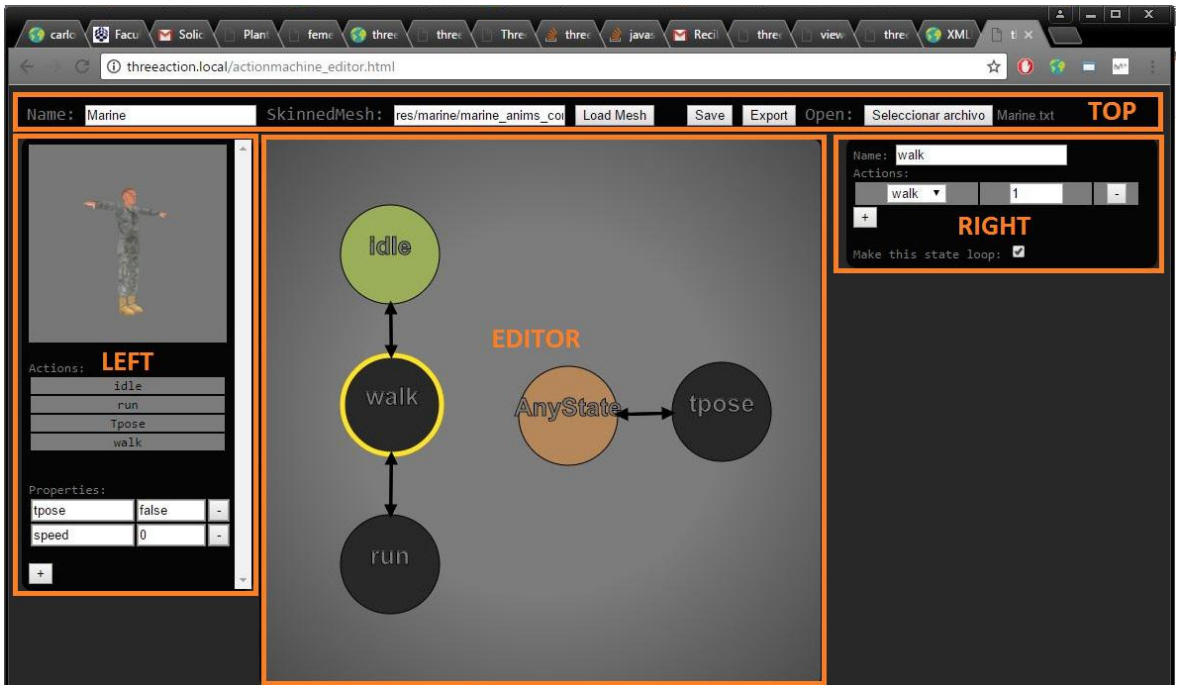


Figura B-1: Secciones de la interfaz

El editor está dividido en 4 secciones principales:

- **Top:** En esta sección se encuentran las opciones más relaccionadas con aspectos generales como puede ser el nombre o las opciones de guardado y cargado.
- **Pannel Left:** En esta parte encontramos el previsualizador de referencia y una lista con las acciones disponibles para ese modelo. También se encuentra el editor de propiedades que permite añadir, eliminar y modificar las propiedades.
- **Editor:** Es el editor gráfico y la parte central de la interfaz. Permite interactuar directamente con la ActionMachine que se está creando.
- **Pannel Right:** Este menú cambiará en función del elemento que se esté seleccionando y permite configuarlo.

En las siguientes secciones entraremos más en detalle en cada una de estas secciones.

B.1 Top

Las principales acciones que nos permite realizar este pannel son las siguientes (de izquierda a derecha):

- **Name:** TextBox en la que se inserta el nombre que se desee dar a la ActionMachine.

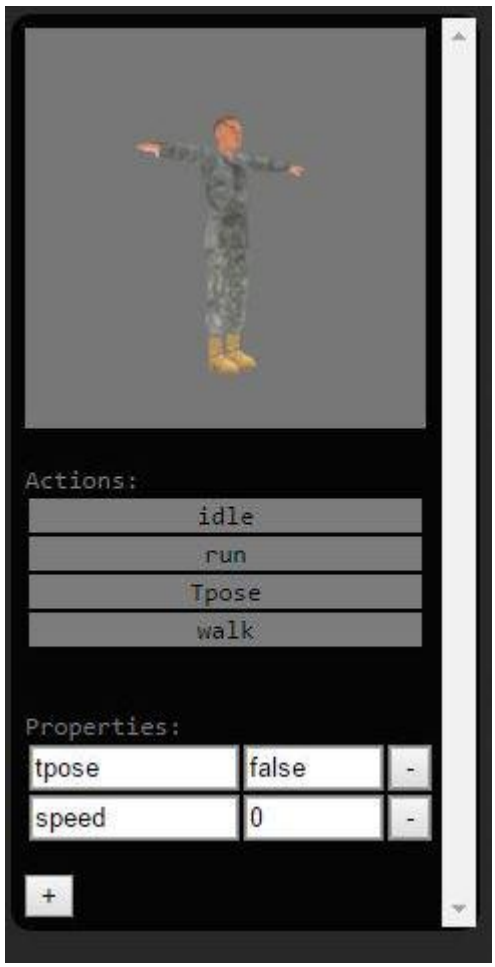
- **SkinnedMesh:** Directorio relativo al editor donde se encuentre el skinned mesh deseado. Es importante que se encuentre en un directorio con permisos de apache. Una vez se ponga el directorio hay que pulsar el botón **Load Mesh.**

- **Save:** Descarga un fichero que contiene el estado actual del editor y toda la información sobre la máquina de estados que se está generando.

- **Export:** Descarga el fichero en formato .js listo para integrar en el código.

- **Open:** Permite cargar un fichero descargado con el botón **Save** para continuar un trabajo en progreso.

B.2 Pannel Left



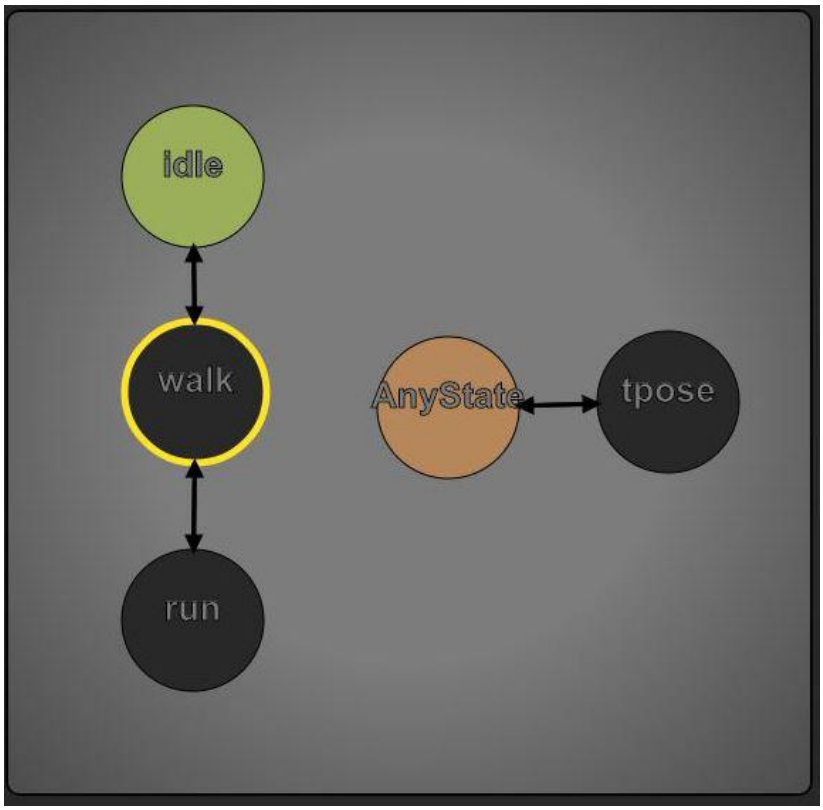
Vamos a desgranar el pannel de la izquierda en cada una de sus secciones de arriba a abajo:

- **Previsualizador del modelo:** Muestra el SkinnedMesh cargado de forma estática. Su finalidad es sencillamente de referencia.

- **Actions:** Lista de acciones o animaciones disponibles en ese modelo.

- **Properties:** Lista de propiedades de la ActionMachine. Se puede modificar el nombre, el valor inicial. También se permite insertar o eliminar todas las que sean necesarias.

B.3 Editor



Como ya hemos explicado el editor gráfico es la parte central y más importante de este editor. Aquí describiremos las acciones que se pueden realizar con él:

- **Drag & Drop:** Con un gesto sencillo de drag & drop se pueden mover los estados y la cámara

- **Zoom:** Haciendo uso de la rueda del ratón se puede hacer zoomIn y zoomOut

- **Doble click:** Haciendo doble click sobre cualquier estado o en la flecha de una transición se seleccionará el elemento resaltándose en amarillo

- **SHIFT + doble click:** haciendo doble click en cualquier parte del lienzo donde no haya un estado mientras se mantiene pulsada la tecla SHIFT se creará un estado nuevo.

- **SHIFT + Drag & Drop:** si mientras se mantiene la tecla shift se incia un movimiento de Drag & Drop desde un estado existente se creará una nueva transición entre este estado y el estado sobre el que se suelte el ratón. Si se suelta el ratón en un espacio vacío se creará un nuevo estado objetivo.

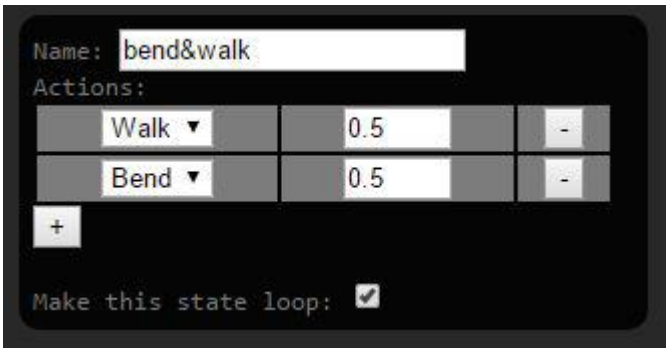
- **DEL:** Cuando se pulse la tecla DEL se eliminarán los elementos seleccionados.

B.4 Pannel Right

Este panel es el que nos permite configurar los elementos del ActionMachine. Cuando se selecciona un estado o una transición se mostrarán sus propiedades y se podrán modificar según las necesidades del diseño.

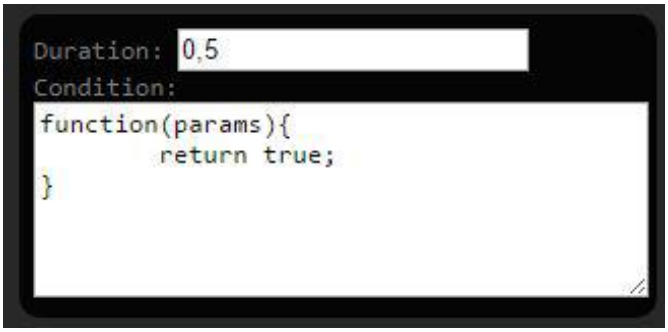
Por ello, este panel mostrará unas propiedades distintas cuando tengamos seleccionado un estado y cuando tengamos seleccionada una transición

Cuando se tiene seleccionado un estado podremos modificar:



- **Name:** Nombre del estado
- **Actions:** Lista de acciones y pesos del BlendTree
- **Loop:** Se deja marcado si se desea que la animación se repita indefinidamente

Cuando lo que se selecciona es una transición, el panel de la izquierda se verá como sigue:



- **Duration:** Tiempo que dura la transición
- **Condition:** Función javascript que recibe params (las propiedades de la máquina de estados en formato Object) como parámetro y devuelve un valor true si se tiene que activar la transición y false si no.

C Manual del programador. Guía de integración.

Este apéndice está destinado al programador de un videojuego en Three.js que desee integrar ThreeAction en su proyecto.

```
<script src="js/threejs/three.js"></script>
<script src="js/ActionMachine.js"></script>

<script src="js/machines/MyActionMachine1.js"></script>
<script src="js/machines/MyActionMachine2.js"></script>
...
<script src="js/machines/MyActionMachineN.js"></script>
```

La primera parte será insertar el directorio threeAction en el proyecto. En él se encuentra todo el código fuente necesario para integrar el motor.

Es importante insertar los archivos en este orden para que se resuelvan correctamente las dependencias.

Los ficheros MyActionMachineN.js representan las máquinas de acciones que quieras incluir en tu juego. Si has generado el código con el editor, cada uno de estos ficheros definirá una función que instancia la máquina de estados.

Por tanto, para usar la action machine deberás inicializarla (en la función start, init o donde se carguen los recursos del juego) con un comando como el siguiente:

```
// Instanciado de un ActionMachine
MyActionMachine1 = _MyActionMachine1_()
```

A partir de este punto la maquina funciona como cualquier objeto de Three.js. Habrá que añadirlo a la escena de la forma habitual, integrar la lógica que se quiera utilizar para moverlo interactuar con el juego.

```
// Insertar el personaje en la escena
scene.add( MyActionMachine1);
```

Cuando se quieran realizar cambios en la máquina de estados habrá que actualizar las propiedades del elemento. Hay que asegurarse de que en cada iteración del juego estas se actualicen correctamente para garantizar una correcta interacción del juego con las animaciones:

```
// Modificar las propiedades de la máquina
MyActionMachine1.setProperty("speed", speed.mod);

// Recuperar las propiedades de la máquina
Speed = MyActionMachine1.getProperty("speed");
```

Por último es necesario llamar todos los ciclos de ejecución a la función update(deltaTime) que actualiza las animaciones.

```
var delta = clock.getDelta();

MyActionMachine1.update( delta );
```

C.1 Directorio load

Cabe la posibilidad de que tras haber generado el código y al mover los archivos de un lugar para otro los modelos no se carguen de forma correcta. Esto es porque el código que se genera carga el objeto con una dirección relativa al editor. Si el código html principal del juego que estás desarrollando se encuentra en un directorio distinto al del editor, deberás modificar manualmente estas direcciones.

En el fichero `MyActionMachine1.js` deberás modificar donde pone:

El directorio por la dirección válida, relativa al html principal sobre el que se ejecute el juego.

```
machine.load("res/.../MyModel1.json", start );
```